DAG-Calculus: A Calculus for Parallel Computation

Umut Acar
Carnegie Mellon University

Arthur Charguéraud
Inria & LRI Université Paris Sud, CNRS

Mike Rainey
Inria

Filip Sieczkowski
Inria

Gallium Seminar
February 2016

Parallel Computation

- Crucial for efficiency in the age of multicores
- Many different modes of use and language constructs
- Stress generally on efficiency, semantics not as well understood

Fibonacci with fork-join

```
function fib (n)
  if n <= 1 then n else
    let (x,y) = forkjoin (fib (n-1), fib (n-2)) in
    x + y
```

1. Perform two recursive calls in parallel

2. Evaluate the result

Fibonacci with async-finish

```
function fib (n)
  if n <= 1 then n else
   let (x,y) = (ref 0, ref 0) in
   finish { async (x := fib (n-1));
            async (y := fib (n-2)) };
   !x + !y
```

1. Perform two recursive calls in parallel

2. Synchronise on completion of parallel calls

3. Read results of calls and compute final result

Fibonacci with futures

```
function fib (n)
  if n <= 1 then n else
    let (x,y) = (future fib (n-1), future fib (n-2))
    in (force x) + (force y)
```

1. Perform two recursive calls in parallel

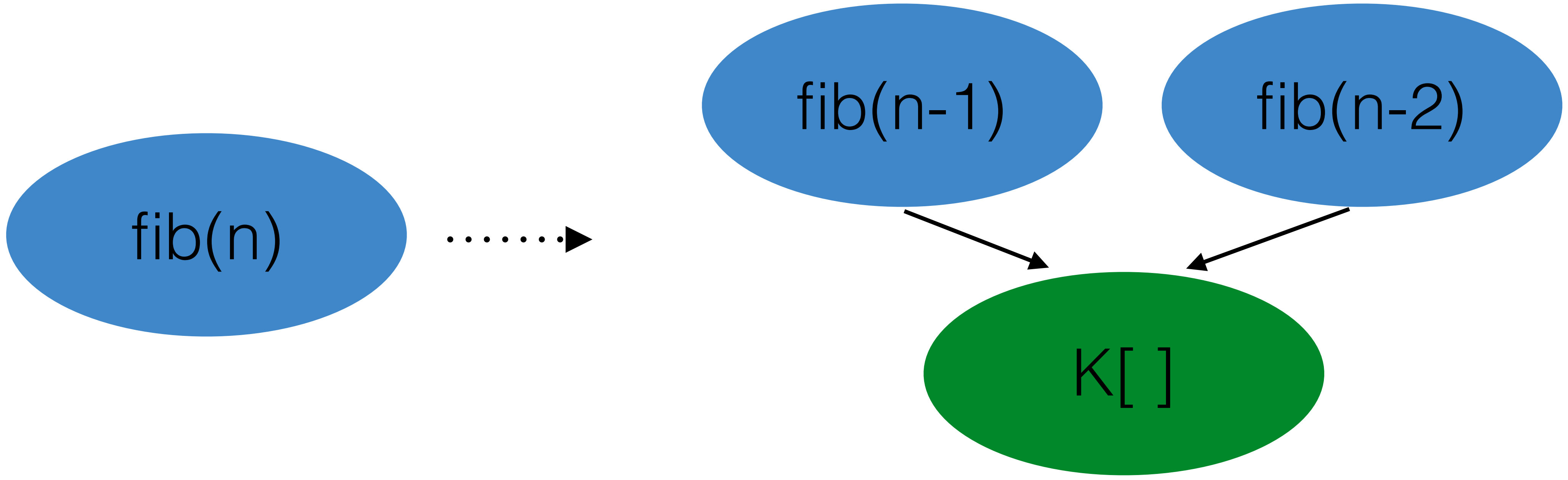
2 & 3. Demand results from the parallel calls, and evaluate the final result

Motivating questions

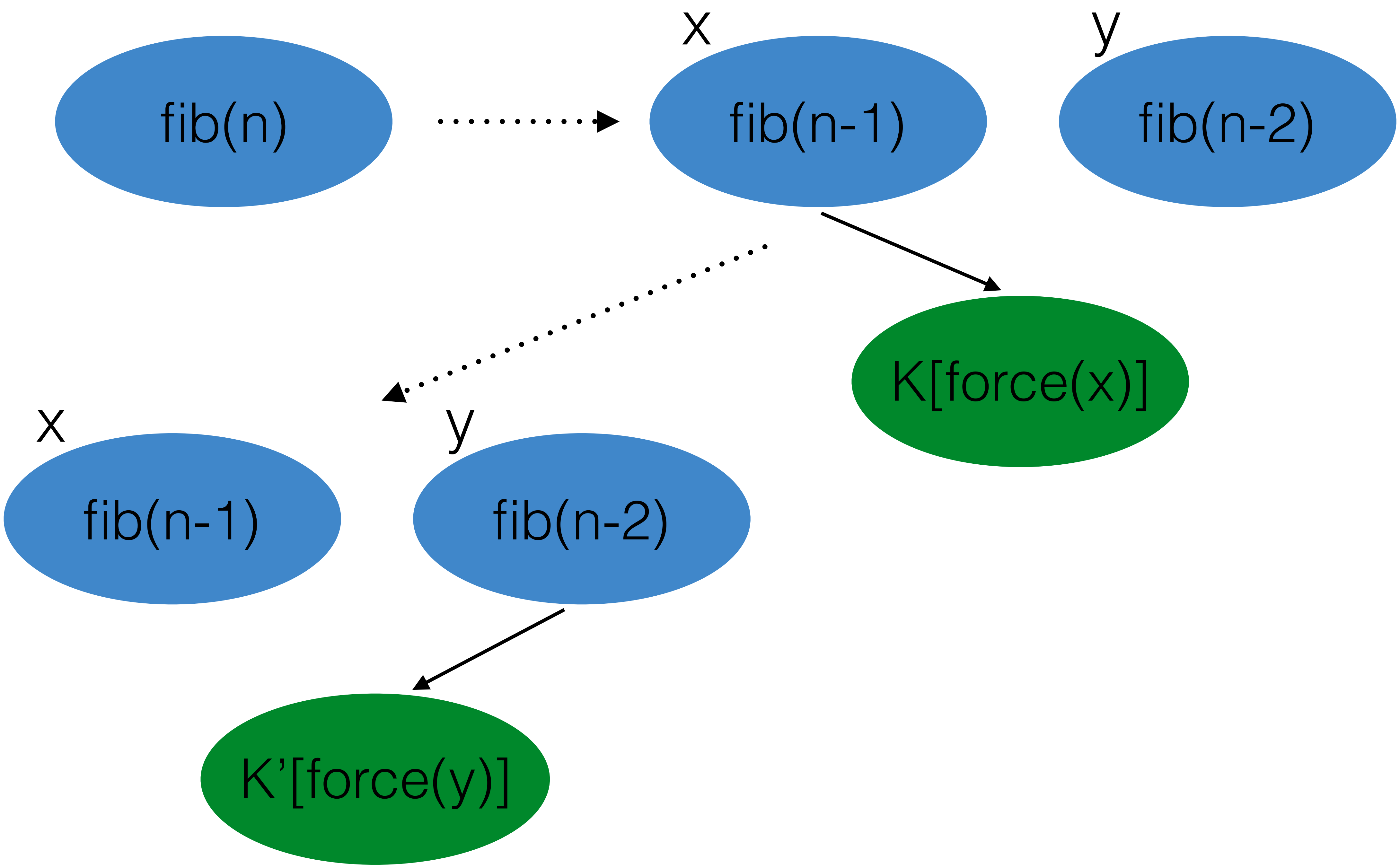
Is there a unifying model or calculus that can be used to express and study different forms of parallelism?

Can such a calculus be realised efficiently in practice as a programming language, which can serve, for example, as a target for compiling different forms of parallelism?

Parallelism patterns: fork-join, async-finish



Parallelism patterns: futures



Core idea — reify the dependency edges

Computing with DAGs

- State of computation: (V, E, σ)
- V — a set of DAG vertices, each with associated *program* and *status*
- E — a set of DAG edges
- σ — shared mutable state
- Side conditions enforce edges form DAG, status of vertices, etc.

Manipulating the DAG

- Four commands that are used to dynamically modify the computation DAG
- `newTd(e)` creates a new vertex in the DAG
- `newEdge(e, e’)` creates an edge from `e` to `e’`
- `release(e)` is called to mark that the vertex `e` is now set up and can be scheduled
- `transfer(e)` is a parallel control operator that transfers the outgoing edges of calling thread to `e`

The life-cycle of a vertex

- Vertex can have one of four status values: New, Released, eXecuting or Finished (N, R, X, F)
- Node created by running **`newTd`** has status N
- After calling **`release`**, its status is changed to R
- At this point it can be *scheduled* for execution, which sets the status to X
- After the execution terminates, the status is set to F

Operational Semantics (1)

$$\frac{V(t) = (K[\texttt{newTd}\ e], \mathsf{X}) \qquad t' \ \text{fresh}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[t'], \mathsf{X})][t' \mapsto (e, \mathsf{N})], E, \sigma}\ \textsc{NewTd}$$

$$\frac{\begin{array}{c} V(t) = (K[\texttt{release}\ t'], \mathsf{X}) \\ V(t') = (e, \mathsf{N}) \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[\texttt{()}], \mathsf{X})][t' \mapsto (e, \mathsf{R})], E, \sigma}\ \textsc{Release}$$

$$\frac{V(t) = (e, \mathsf{R}) \qquad \{t' \mid (t', t) \in E\} = \emptyset}{V, E, \sigma \twoheadrightarrow V[t \mapsto (e, \mathsf{X})], E, \sigma}\ \textsc{Start}$$

$$\frac{V(t) = (e_1, \mathsf{X}) \qquad \sigma_1, e_1 \to e_2, \sigma_2}{V, E, \sigma_1 \twoheadrightarrow V[t \mapsto (e_2, \mathsf{X})], E, \sigma_2}\ \textsc{Step}$$

$$\frac{V(t) = (v, \mathsf{X}) \qquad E' = E \setminus \{(t, t') \mid t' \in \mathrm{dom}(V)\}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (\texttt{()}, \mathsf{F})], E', \sigma}\ \textsc{Stop}$$

Operational Semantics (2)

$$\frac{\begin{array}{c} V(t) = (K[\texttt{newEdge}\ t_1\ t_2], \mathsf{X}) \qquad t_1, t_2 \in \mathrm{dom}(V) \\ \mathrm{status}(V(t_2)) \in \{\mathsf{N}, \mathsf{R}\} \qquad E' \text{ cycle-free} \\ E' \;=\; E \;\uplus\; (\text{if status}(V(t_1)) = \mathsf{F} \text{ then } \emptyset \text{ else } \{(t_1, t_2)\}) \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[\texttt{()}], \mathsf{X})], E', \sigma}\ \textsc{NewEdge}$$

$$\frac{\begin{array}{c} V(t) = (K[\texttt{transfer}\ t'], \mathsf{X}) \qquad \mathrm{status}(V(t')) = \mathsf{N} \qquad \{t'' \mid (t', t'') \in E\} = \emptyset \\ E' \;=\; E \;\setminus\; \{(t, t'') \mid t'' \in \mathrm{dom}(V)\} \;\uplus\; \{(t', t'') \mid (t, t'') \in E\} \qquad E' \text{ cycle-free} \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[\texttt{()}], \mathsf{X})], E', \sigma}\ \textsc{Transfer}$$

Encoding fork-join

```
⟦forkjoin (e1, e2)⟧ =
  capture (fn k =>
    let l1 = alloc
        l2 = alloc

        t1 = newTd (l1 := ⟦e1⟧ )
        t2 = newTd (l2 := ⟦e2⟧ )
        t  = newTd (k (!l1, !l2))

    in newEdge(t1, t); newEdge(t2, t)
       transfer(t); release(t);
       release(t1); release(t2))
```

Encoding async-finish

```
⟦t | async(e)⟧ =
   let t’ = newTd( ⟦t | e⟧ )
   in newEdge(t’, t); release(t’)
```

```
⟦t | finish(e)⟧ =
   capture (fn k =>
     let t2 = newTd(k ())
         t1 = newTd( ⟦t2 | e⟧ )
     in newEdge(t1, t2); transfer(t2);
        release(t2); release(t1))
```

Encoding futures

```
⟦future(e)⟧ =
   let l  = alloc
       t  = newTd(())
       t’ = newTd(l := ⟦e⟧ )
   in newEdge(t’, t); release(t);
      release(t’); (t, l)
```

```
⟦force(e)⟧ =
   let (t, l) = ⟦e⟧
   in capture (fn k =>
      let t’ = newTd (k (!l))
      in newEdge(t, t’);
         transfer(t’); release(t’)
```

Proving the encodings correct: the technique

- Compiler-style proof of simulation
- Need *backwards*-simulation due to nondeterminism
- Problem: partially evaluated encodings do not correspond to any source terms
- Solution: an intermediate, annotated language, two-step proof
- Keep the structure *and* allow partial evaluation of parallel primitives

Also in the paper

- Scheduling DAG-calculus computations using work stealing
- Data-structures for efficient implementation of DAG edges
- Experimental evaluation of implementation

Conclusions

- A unifying calculus: common framework for expressing different modes of parallelism
- A low-level calculus: useful as an intermediate language/mental model rather than directly
- An efficient implementation using novel data-structures to handle high-degree vertices